算法设计与分析大作业报告

邵宇鹏

2024 年 6 月 22 日

1 问题一：简述分治策略的思想

分治策略是一种重要的算法设计方法，其核心思想是将一个复杂的问题分解成若干个规模较小但类似于原问题的子问题，递归地求解这些子问题，然后将这些子问题的解组合成原问题的解。这种方法通过”分而治之”的方式，将难以直接解决的问题转化为易于处理的形式。

1.1 分治策略的基本步骤

分治策略通常包含以下三个主要步骤：

1. **分解（Divide）**：将原问题分解为若干个规模较小、相互独立、与原问题形式相同或类似的子问题。
2. **解决（Conquer）**：递归地求解各个子问题。如果子问题规模足够小，则直接求解。
3. **合并（Combine）**：将各个子问题的解合并成原问题的解。

1.2 分治策略的特点

- **递归结构**：分治算法通常使用递归来实现，每一级递归都会将问题进一步分解。

- **子问题独立**：各个子问题之间相互独立，不包含公共的子子问题。
- **问题规模缩小**：通过分解，子问题的规模比原问题显著减小，最终达到易于直接求解的程度。
- **问题同质**：子问题与原问题属于同一类型，可以用相同的方法求解。

1.3 分治策略的应用实例

为了更好地理解分治策略，我们可以通过一些经典的算法实例来说明：

1. **归并排序**：将待排序序列分成两半，分别对两部分进行排序，然后将排好序的两部分合并。
2. **快速排序**：选择一个基准元素，将序列分为小于基准和大于基准的两部分，分别对这两部分进行排序。
3. **大整数乘法**：将大整数分成位数较少的部分，分别相乘后再合并结果。
4. **棋盘覆盖问题**：将棋盘分成四个小棋盘，分别解决覆盖问题，然后合并解决方案。

1.4 分治策略的优势与局限性

优势：

- 能有效地解决大规模复杂问题
- 易于并行实现，提高计算效率
- 对于一些问题，可以显著降低时间复杂度

局限性：

- 可能产生大量的重复计算（这种情况下，动态规划可能更合适）

- 递归调用可能导致较大的栈空间开销
- 不是所有问题都适合使用分治策略

总的来说，分治策略是一种强大的算法设计工具，在计算机科学中有着广泛的应用。它通过将复杂问题分解为更小、更易管理的部分，为许多重要算法提供了基础，如排序、搜索、最优化等领域的经典算法。理解和掌握分治策略对于算法设计和问题解决能力的提升至关重要。

2 问题二：简述动态规划的思想

动态规划（Dynamic Programming，简称 DP）是一种将复杂问题分解成更简单的子问题来解决的优化技术。它的核心思想是通过存储中间结果来避免重复计算，从而提高算法的效率。动态规划通常用于解决具有重叠子问题和最优子结构性质的问题。

2.1 动态规划的基本原理

动态规划的基本原理包括以下几个关键点：

1. **最优子结构**：问题的最优解包含子问题的最优解。
2. **重叠子问题**：在求解过程中，很多子问题会重复出现。
3. **状态转移**：定义问题状态，并找出状态之间的转移关系。
4. **边界条件**：确定初始状态或最小子问题的解。
5. **空间换时间**：使用额外的存储空间来记录子问题的解，避免重复计算。

2.2 动态规划的实现方法

动态规划主要有两种实现方法：

- **自顶向下（记忆化搜索）**：从原问题出发，递归地解决子问题，并将子问题的解存储起来。
- **自底向上（迭代）**：从最小的子问题开始，逐步解决较大的子问题，直到解决原问题。

2.3 动态规划的步骤

解决动态规划问题通常遵循以下步骤：

1. **定义状态**：明确定义问题的状态，通常用一个或多个变量表示。
2. **确定状态转移方程**：找出状态之间的关系，用数学表达式表示。
3. **确定边界条件和初始状态**：明确最小子问题的解和初始条件。
4. **确定计算顺序**：决定如何遍历状态空间以保证每个子问题在使用前已被解决。
5. **实现算法**：根据以上步骤，编写代码实现动态规划算法。

2.4 动态规划的典型应用

动态规划在许多领域都有广泛应用，一些经典问题包括：

- **斐波那契数列**：使用动态规划可以在 O(n) 时间内计算第 n 个斐波那契数。
- **最长公共子序列（LCS）**：寻找两个序列中最长的公共子序列。
- **背包问题**：在给定容量限制下，选择物品以最大化总价值。
- **最短路径问题**：如 Floyd-Warshall 算法，用于寻找图中所有点对之间的最短路径。
- **矩阵链乘法**：寻找最优的矩阵乘法顺序以最小化计算量。

2.5 动态规划的优势与局限性

优势：

- 可以有效解决具有重叠子问题的复杂问题
- 通常比暴力递归更高效
- 适用于寻找最优解的问题

局限性：

- 需要额外的存储空间来保存中间结果
- 有时难以找到正确的状态表示和状态转移方程
- 不是所有问题都适合使用动态规划（如不具有最优子结构的问题）

总之，动态规划是一种强大的问题解决技术，特别适合于具有重叠子问题和最优子结构的优化问题。它通过巧妙地利用问题的结构，避免重复计算，从而大大提高了算法的效率。掌握动态规划的思想和技巧对于解决复杂的算法问题至关重要，是每个程序员和算法设计者必备的技能之一。

3 问题三：简述贪心法的设计思想

贪心算法 (Greedy Algorithm) 是一种在每一步选择中都采取在当前状态下最好或最优 (即最有利) 的选择, 从而希望导致结果是全局最好或最优的算法。

3.1 贪心法的核心思想

贪心算法的核心思想可以概括为:

- 问题可以分解成若干个子问题

- 每个子问题的最优解可以递推到最终问题的最优解
- 每次决策时都采取当前状态下最优的选择

3.2 贪心法的设计步骤

设计一个贪心算法, 通常遵循以下步骤:

1. 将问题分解为一系列子问题
2. 对每个子问题, 定义一个期望的贪心选择
3. 证明每次贪心选择都是安全的, 即局部最优选择能导致全局最优解
4. 证明所有子问题的贪心选择组合起来能得到原问题的最优解

3.3 贪心法的特点

贪心算法具有以下特点:

- 简单, 直观
- 通常情况下运行效率高
- 对于某些问题能得到最优解
- 不能保证对所有问题都能得到最优解

3.4 贪心法的应用场景

贪心算法适用于具有”贪心选择性质”的问题, 即局部最优选择能导致全局最优解。常见的应用场景包括:

- 最小生成树问题 (如 Kruskal 算法、Prim 算法)
- 单源最短路径问题 (如 Dijkstra 算法)

- 任务调度问题
- 哈夫曼编码

总之, 贪心法是一种通过局部最优选择, 试图达到全局最优解的算法设计方法。它在许多问题中能够快速有效地得到最优解或近似最优解, 是一种重要的算法设计思想。

4 问题四：简述回溯算法的设计思想

回溯算法 (Backtracking) 是一种通过试错的方式来寻找问题解的算法。它尝试分步地构造候选解, 如果发现当前候选解不可能导致有效的解决方案, 就” 回溯” 到上一步, 尝试另一种选择。

4.1 回溯算法的核心思想

回溯算法的核心思想可以概括为:

- 将问题的解空间组织成树或图的形式
- 深度优先搜索整个解空间
- 在搜索过程中, 根据约束条件剪枝, 避免无效搜索
- 找到解或确定无解后回溯

4.2 回溯算法的设计步骤

设计一个回溯算法, 通常遵循以下步骤:

1. 确定问题的解空间
2. 确定易于递归实现的搜索路径

3. 确定剪枝条件

4. 设计递归函数, 实现搜索和回溯

4.3 回溯算法的特点

回溯算法具有以下特点:

- 适用于求解组合优化问题
- 可以找到所有可能的解
- 时间复杂度通常较高, 但通过剪枝可以显著改善效率
- 空间复杂度相对较低, 主要是递归调用栈的开销

4.4 回溯算法的应用场景

回溯算法广泛应用于以下场景:

- 排列组合问题 (如 N 皇后问题)
- 图的着色问题
- 数独求解
- 0-1 背包问题
- 旅行商问题

4.5 回溯与深度优先搜索的关系

回溯算法可以看作是带有剪枝的深度优先搜索 (DFS)。主要区别在于:

- 回溯算法在搜索过程中会进行剪枝

- 回溯算法不仅关注是否有解, 还要找出所有可能的解
- 回溯算法通常用于求解优化问题或可行性问题

总之, 回溯算法是一种通过系统地尝试各种可能性来找到所有解或最优解的方法。它在组合优化问题中特别有用, 通过合理的剪枝可以有效减少搜索空间, 提高算法效率。

5 问题五

5.1 问题描述

设有 n 种不同面值的硬币，第 i 种硬币的面值是 v_i，重量是 w_i（$i = 1, 2, \ldots, n$）。现需要支付总价值为 y 的商品，若每种硬币使用的个数不限，问如何选择这些硬币使得付出的总重量最轻？

5.2 数学描述

我们的问题是找到一种方法，用最轻的总重量支付总价值为 y 的商品。具体地，我们有 n 种不同面值的硬币，每种硬币的面值分别为 $v_1, v_2, \ldots, v_n$，相应的重量分别为 $w_1, w_2, \ldots, w_n$。我们的目标是使支付总价值 y 的硬币组合的总重量最小。

定义一个数组 $dp[j]$，表示支付总价值为 j 时的最小重量。我们将 $dp[0]$ 初始化为 0，表示支付总价值为 0 时所需的重量为 0。对于其他的 j 值，初始设定为无穷大，表示尚未计算出可行的方案。

我们的递推方程为：

$$dp[j] = \min\left(dp[j], dp[j - v_i] + w_i\right) \quad \text{对于每个硬币 } i \text{ 和 } v_i \leq j$$

这表示对于每个硬币面值 v_i 和相应的重量 w_i，如果当前总价值 j 可以通过使用该硬币来实现（即 $j \geq v_i$），那么我们可以选择是否使用该硬币来更新 $dp[j]$ 的值。

算法步骤

1. 初始化一个长度为 $y+1$ 的数组 dp，所有元素值设为无穷大，除了 $dp[0]$ 设为 0。
2. 对于每一个面值的硬币 v_i：
   - 遍历从 v_i 到 y 的所有总价值 j：
     - 更新 $dp[j]$ 为 $\min(dp[j], dp[j-v_i]+w_i)$，表示是否选择使用当前硬币来达到总价值 j。
3. 最后，$dp[y]$ 的值即为支付总价值 y 所需的最小总重量。

考虑以下输入：

- 面值：$v_1=1, v_2=4, v_3=6, v_4=8$
- 重量：$w_1=1, w_2=2, w_3=4, w_4=6$
- 总价值：$y=12$

使用上述算法，我们可以得到支付总价值 12 的最小总重量。

5.3 算法设计

本问题可以使用动态规划的方法来解决。动态规划的核心是将大问题划分为小问题，通过解决小问题来推导大问题的解。以下是算法的伪代码。

Algorithm 1 Min Weight Algorithm

输入: n 种硬币的面值 $values[1 \ldots n]$ 和重量 $weights[1 \ldots n]$，需要支付的总价值 y
输出: 支付总价值为 y 的最小重量
初始化 dp 数组，长度为 $y+1$，所有值设置为正无穷大，$dp[0]$ 设为 0
for $i = 1$ **to** n **do**
  for $j = values[i]$ **to** y **do**
    $dp[j] = \min(dp[j], dp[j - values[i]] + weights[i])$
  end for
end for
return $dp[y]$

5.4 MATLAB 实现

以下是上述算法在 MATLAB 中的实现代码。

Listing 1: MATLAB 代码

```
function min_weight = minWeight(n, values, weights, y)
    % 初始化 dp 数组
    dp = inf(1, y+1);
    dp(1) = 0;

    % 动态规划填表
    for i = 1:n
        for j = values(i):y
            dp(j+1) = min(dp(j+1), dp(j-values(i)+1) + weights(i));
        end
    end

    % 返回 dp(y+1)
```

```
    min_weight = dp(y+1);
end

% 输入示例
values = [1, 4, 6, 8];
weights = [1, 2, 4, 6];
y = 12;

% 调用函数并输出结果
result = minWeight(4, values, weights, y);
fprintf('最小总重量: %d\n', result);
```

5.5 运行结果

对于输入 $v_1 = 1, v_2 = 4, v_3 = 6, v_4 = 8$ 和 $w_1 = 1, w_2 = 2, w_3 = 4, w_4 = 6$，需要支付总价值 $y = 12$ 时，运行上述算法可以得到最小的总重量。具体地，通过动态规划算法，我们得到支付总价值为 12 时的最小总重量为 6。

5.6 结论

本文通过动态规划的方法解决了不同面值和重量的硬币支付问题。通过递推关系，我们可以高效地计算出支付指定金额所需的最小重量。这一方法在实际应用中具有广泛的应用前景，可以用于优化支付系统中的重量问题。